animation

background

The word animation is a mideval term stemming from the Latin animare, which means ‘instill with life’. In modern terms, it’s used to describe the process of creating movement from still, sequential images. Early creators of animation used spinning discs (phenakistoscope) and cylinders (zoetrope) with successive frames to create the illusion of a smooth movement from persistance of vision. In modern times, we’re quite used to other techniques such as flip books and cinmetic techniques like stop motion. Increasingly, artist have been using computational techniques to create animtion – using code to “bring life” to objects on the screen over successive frames. This chapter is going to look at these techniques and specifically, try to address a central question, “how can we create compelling, organic and absurd movement through code”

As a side note, I studied fine arts, painting and printmaking, and it was accidental that I started using computers. The moment that I saw how you could write code to move something across the screen, even as simple as silly rectangle, I was hooked. I began during the first dot com era working with flash / actionscript and lingo / director and have never looked back.

This chapter will first explain some basic principles that are useful to understanding animation in OF, then attempt to show a few entrypoints to intersting approaches.

animation in OF / useful concepts:

draw cycle

The first point to make about animation is that it’s based on successive still frames. In openFrameworks we have a certain loop cycle that’s based roughly on game programming paradigms. It goes like:

- setup()
- update()
- draw()
- update()
- draw()
- ....

setup gets called once and update / draw get called repeatedly. Sometimes people ask why two functions the called repeatedly? Especially if they are used to processing, which has only a setup and a draw command. There are a few reasons. The first is that drawing in opengl is asynchronous, meaning there's a chance, when you send drawing code to the computer, that it can return execution back to your program so that it can do other operations while it draws. The second is that it's generally very practical to have your drawing code seperated from your non drawing code. If you need to quickly debug something – say for example your code is running slow – you can comment out the draw function and just leave the update running. It's seperating out the update of the world from the presentation and it can often help clean up / organize your code. Mentally think about it like an stop frame animator working with an overhead camera who might reposition objects while the camera is not taking a picture then snap a photograph the moment things are ready. In the update function you would be moving things around and in the draw function you draw things exactly as they are at that moment.

variables

The second point to make about animation is that it requieres variables. A variable is a placeholder for a value, which means that you can put the value in and you can also get the value out. Variable are essential for animation since they "hold" value from frame to frame. Ie, if you put a value in to a variable in the setup function or update function, you can also get it out from memory in the draw function. Take this example:

[**note: simple animation example here**]

frame rate

The third point to make about OF and animation is frame rate. We animate in openframeworks using successive frames. Frame rate refers to how quickly frames get drawn. In OF we have several important functions to know about.

- `ofGetFrameRate()` returns the current frame rate (in frames per second). Set it 0 to run as fast as possible [**note:double check this**]
- `ofSetFrameRate( float targetFrameRate )` sets the maximum frame rate. If the software is animating faster then this, it will slow it down. Think of it like a speed limit. It doesn't make you go faster, but it prevents you from going to fast.

In addition, openGL works with an output display and will attempt to synchronize with the refresh rate of the monitor – sometimes called vertical-sync or vertical blanking. If you don't synchronize with the refresh rate, you can get something called frame tearing, where the non synchronization can mean frames get drawn before and after a change, leading to horizontal lines of discontinuity.

[**note: frame rip graphic here**]

We have a function in OF for controlling this. Some graphics cards drivers (see for example Nvidia's PC drivers) have settings that over-ride application settings, so please be sure to check your driver options.

- `ofSetVerticalSync( bool bUseSync )` set this true if you want to synchronized vertically, false if you want to draw as fast as possible.

By default, OF enables vertical sync and sets a frame rate of 60FPS. You can adjust the VSYCN and frame rate settings if you want to animate faster, but please note that by default OF wants to run as fast as possible. It's not uncommon if you are drawing a simple scene to see frame rates of 800 FPS if you don't have VSYNC enabled (and the frame rate cap set really high or disabled).

[**note: discuss frame rate independence?**]

time functions

Finally, there are a few functions that are useful generally for knowing about timing:

- `ofGetElapsedTimef()` retuns the elapsed time in floating point numbers, starting from 0 when the app starts.
- `ofGetElapsedTimeMillis()` similarly returns the elapsed time starting from 0 in milliseconds
- `ofGetFrameNum()` returns the number of frames the software has drawn. If you wanted, for example, to do something every other frame you could use the mod operator, ie, `if (ofGetFrameNum() % 2 == 0)`.

objects

In these examples, I'll be using objects pretty heavily. It's sort of helpful to feel comfortable with OOP to understand the code. One object that is used really heavily is `ofPoint`, which essentially contains an x,y and a z variable. In the past this was called ofVec3f (vector of three floating point numbers) but we just use the more convenient ofPoint. In some animation code you'll see vectors used, you should know that ofPoint is essentially a vector.

You will also typically see objects that have basic functionality and internal variables. I will typically have a setup, update and draw inside them. Alot of times, these objects are either made because they are useful recipies to have many things on the screen or they help by putting all the variables and logic of movement in one place. I typically like to have as little code as possible at the testApp / ofApp level. If you are familiar with actionscript / flash, this would be similar to having as a little as possible in your main timeline.

linear movement

getting from point a to point b

One of the most important things to think about when it comes to animation is answering the simple question:

how do you get from point a to point b.

For the most part in this chapter we will look at animating movement (changing position over time) but we very well could be animating any other numeric property, such as color, the width or height of a drawn shape, radius of a circle, etc.

The first and probably most important lesson of animation is that we **love** numbers between 0 and 1.

[**note: love picture here**]

The thing about numbers between 0 and 1 is that they are super easy to use in interesting ways. We typically refer to these kinds of numbers as percent, and you'll see me use the shorthand `pct` in the code – this is a floting point number between 0 and 1. If we wanted to get from point A to point B, we could use this number to figure out how much of one point and how much of another point to use. The formula is this:

```
((1-pct) * A) + (pct * B)
```

to add some detail if we are 0 pct of the way from A to B, we calculate:

```
((1-0) * A) + (0 * B)
```

which simplifies to `(1*A + 0*B)` or A. If we are 25 percent of the way, it looks like:

```
((1-0.75) * A) + (0.25 * B)
```

which is 75% of A + 25% of B. Essentially by taking a mix, you get from one to the other. The first example shows how this is done.

[**note: linear example code here**]

(as a side note, the function `ofMap`, which maps between an input range, uses pct internally. it takes a value, converts it into a percentage based on the input range, and then uses that pct to find the point between the output range) [**note: see omer's chapter**]

curves

One of the interesting properties of numbers between 0 and 1 is that they can be easily adjusted / curved.

The easiest way to see this is by raising the number to a power. a power, as you might remember from math class, is multiplying a number by itself. ie, 2ˆ3 [**note: latex helpful here?**] = `2*2*2 = 8`. Numbers between 0 and 1 have some interesting properties – if you raise 0 to any power it equals 0 (`0x0x0x0 = 0`). The same thing is true for 1 (`1*1*1*1 = 1`). But if you raise a number between 0 and 1 to a power, it changes. 0.5 to the 2nd power = 0.25.

Let's look at a plot of pct raised to the second power:

[**note: plot grpahic here**]

[**note: better explanation of how to read the chart**] Think about the x value of the plot as the input and y value as the output. If put in 0, we get out a y value of 0, if we put in 0.1, we get out a y value of 0.01, all the way to putting in a value of 1 and getting out a value of 1.

The interesting thing is is that things in the world don't move linearly. They don't take even steps. Roll a ball on the floor, it slows down. It's deccellerating. Something things speed up, like a baseball bat going from resting to swinging. Curving pct leads to interesting behavior. The objects still take the same amount of time to get there, but they do it in more lifelike, non-linear ways.

If you raise it to a larger power it looks more extreme. Interestingly, if you raise this value between 0 and 1 to a fractional (rational) power (ie, a power that's less then 1 and greater then 0), it curves in the other direction.

The second example shows an animation which uses pct again to get from A to B, but in this case, pct is raised to a power

http://en.wikipedia.org/wiki/12_basic_principles_of_animation#Slow_in_and_slow_out

[**note: can we get rights for a screenshot of masahiko sato curves DVD ?**]

Raising percent to a power is one of a whole host of functions that are called "shaping functions" or "easing equations". Robert Penner wrote about and derived mand of these functions so they are also commonly reffered to as "Penner Easing Equations." Easings.net is a good resource, as well there are several openframeworks addons for easing.

- http://sol.gfxile.net/interpolation/#c1
- http://easings.net/

zeno

A small twist on the linear interpolation is a technique that I call "zeno" based on zeno the greek philosipher's Dichotomy paradox – if you are running a race and run 1/2 of the remaining distance (getting to 1/2 of the goal) , and 1/2 of the remaining distance (3/4), and 1/2 of the remaining distance (7/8ths)… [**note: better write up**]

If we take the linear interpolation code but instead always alter our own position (ie, take 50% of our current position + 50% of our target position) we can animate our way from one value to another. This is a very easy algorithm to explain by standing up and moving, here's how:

1. Start at one far in your room
2. Pick a point to move to
3. Calculate the distance between your current position and that point
4. Move 50% closer
5. Go to (3)

[**note: diagram for zeno**]

In code, that's basically the same as saying:

```
currentValue = currentValue + ( targetValue - currentValue ) * 0.5;
```

in this case `( targetValue - currentValue )` is the distance. You could also change the size of the step you make every time, for example taking steps of 10% instead of 50%:

```
currentValue = currentValue + ( targetValue - currentValue ) * 0.1;
```

If you expand the expression, you can write the same thing this way:

```
currentValue = currentValue * 0.9 + targetValue * 0.1;
```

this is a form of smoothing – you take some percentage of your current value and another percentage of the target and add them together. Note, those percentages have to add up to 100%, so if you take 95% of the current position, you need to take 5% of the target (ie, `currentValue * 0.95 + target * 0.05`)

In Zeno's paradox, you never actually get to the target, since there's always some remaining distance to go. On the computer, since you are talking about pixel positions on the screen and floating point numbers at a specific range, the object appears to stop.

[**note: code walk through**]

function based movement

In this section of the book we'll look at a few examples that show function based movement, essentially using a function that takes some input and returns an output that we'll animated with. For input, we'll be passing in counters, elapsed time, position, and the output we'll use to control position.

sin cos

Another interesting and simple system to begin experimenting with motion in openframeworks is using sin and cos, two trigonometric functions that are useful inputs into animation.

Sin and cos (sinus and cosinus) are trigonometric functions, which means they are based on angles. They are essentially the x and y positon of a point moving in a constant rate around a circle. The circle is a unit circle, with a radius of 1, which means the diameter is `2*r*PI` or `2*PI`. In OF you'll see this constant as `TWO_PI`, which is 6.28318…

as a side note, sometimes it can be confusing that some functions in OF take degress where others take radians. sin and cos are part of the math library, so they take radians, whereas most opengl rotation takes degrees. We have some helper constants such as `DEG_TO_RAD` and `RAD_TO_DEG` which can help you convert one to the other

all you need to know about sin and cos in one graph So here's a simple drawing

simple examples it's pretty easy to use sin to animate the position of an object.

Here, we'll take the sin of the elapsed time `sin(ofGetElpasedTimef())`. This returns a number between negative one and one. It does this every 6.28 seconds. We can use of map to map this to a new range. For example

```
void ofApp::draw(){
    float xPos = ofMap(sin(ofGetElpasedTimef()), -1, 1, 0, ofGetWidth());
    ofRect(xPos, ofGetHeight/2, 10,10);
}
```

This draws a rectangle which move sinusoidally across the screen, back and forth every 6.28 seconds.

You can do simple things with offseting the phase (how shifted over the sin wave is):

```
void ofApp::draw(){

}
```

nerdy detail! floating points numbers are not not linearly precise, ie, the there's a different number of floating point numbers between 0.0 and 1.0 than 100.0 and 101.0. You actually loose precesion the larger a floating point number gets, so taking sin of elapsed time can start looking crunch after some time. for long running installations I will sometimes write code that looks like `sin((ofGetElapsedTimeMillis() % 6283) / 6283.0)` or something similar, to account for this. Even though ofGetElapsedTime gets larger over time, it's a worse and worse input to sin() as it grows

circular movement Since sin and cos are derived from the circle, if we want to move things in a circular way, we can figure this out via sin and cos. We have 4 variables we need to know:

- the origin of the circle (xOrig,yOrig)
- the radius of the circle (radius)
- the angle around the circle (angle)

The formuala is fairly simple:

```
xPos = xOrig + radius * cos(angle);
yPos = yOrig + radius * sin(angle);
```

This allows us to create something moving in a circular way. For these examples, I start to add a "trail" to the object by using the ofPolyline object. I keep adding points, and once I have a certain number I delete the oldest one. This helps us better see the motion of the object.

If we do things like change the radius of the circle, we can make spirals.

lisajous figures Finally, if we alter the angles we pass in to x and y for this formula in different rates, we can get interesting figures, called "lissajous" figures, named after the french mathematician Jules Antoine Lissajous. These formulas look cool. [**note: more**]

noise

Another useful function for animation is noise. The noise function in OF is based on simplex noise, which is similar to Perlin noise, but requires less calculation. The noise function takes one of two forms:

- `ofNoise()`
- `ofSignedNoise()`

ofNoise returns a number between 0 and 1, and is centered at 0.5, whereas ofSignedNoise returns a number between -1 and 1 and centered at 0.

The most important thing to understand about ofNoise is that it take a various number of input and returns just one output:

```
float ofNoise(float)
float ofNoise(float, float)
float ofNoise(float, float, float)
float ofNoise(float, float, float, float)
```

here's a quick example:

```
void ofApp::draw(){
    float xPos = ofMap(ofNoise(ofGetElpasedTimef()/100.0), -1, 1, 0, ofGetWidth());
    ofRect(xPos, ofGetHeight/2, 10,10);
}
```

Notice the divide by 100.0, think about this as a zoom into the noise. the more you zoom in, the smoother the noise is. The more you zoom out, the higher freuency you see.

here's an example

simulation

If you have a photograph of an object at one point of time, you know it's position. If you have a photograph of an object at another point of time and the camera hasn't changed, you can measure its speed (also referred to as velocity), ie, it's change in distance over time. If you have a photograph at three points in time, you can measure it's acceleration, how much is the speed changing over time.

The individual measures when compared together tell us something about movement. We're actually going to go in the opposite direction now. think about how we can use measures like speed and acceleration to control position.

If you know how fast an object is traveling, you can know how far it is in the certain amount of time. For example, if you are driving at 50 miles per hour (roughly 80km / hour), how far away are you in one hour? Easy right, 50 miles. How far away are you in 2 hours or 3 hours? What you are doing to calculate this is a simple equation:

```
position = position + (velocity * elapsed time)
```

ie:

```
position = position + 50 * 1;   // for one hour away
```

or

```
position = position + 50 * 2;   // for two hours driving
```

The key expression to think about I'll write in shorthand, `p=p+v`, position = position + velocity.

Note, the elapsed time part is important, but when we animate we'll be doing p=p+v quite regularly and you may see us drop this to simplify things (assume every frame has an elapsed time of one). This isn't entirely accurate but it keeps things simple.

In addition, if you are traveling at 50 miles per hour (apologies to everyon who think in km!) and you accelerate by 5 mile per hour, how fast are you driving in 1 hr? 55 mph.... In 2 hrs? 60 mph. Here are you doing the following:

```
velocity = velocity + acceleration
```

In shorthand, I'll write `v=v+a`. So we have two equations we care about for showing movement based on speed:

```
p = p + v;   // position = position + velocity
v = v + a;   // velocity = velocity + acceleration
```

the amazing thing is what we've done here is described a system that can use acceleration to control position. Why is this useful? It's useful because if you remember physics class, Newton had very simple laws of motion, the second of which says:

```
Force = Mass x Acceleration
```

In shorthand, `F = M x A`. This means force and acceleration are linearly related. If we assume that an object has a mass of one, then force equals acceleration. This means we can use force to control velocity and velocity to control position.

The cool, amazing, beautiful thing is that are plenty of forces we can apply to an object, such as spring forces, repulsion forces, alignment forces, etc.

where to go further

physics and animation libraries